March 1993

The Fast Fourier Transforms as a Database Query

Peter Buneman
University of Pennsylvania

The Fast Fourier Transforms as a Database Query

Abstract

By casting the discrete Fourier transform in comprehension syntax and adding some primitives for "array comprehensions", the fast Fourier transform may be derived by direct manipulation of source code.

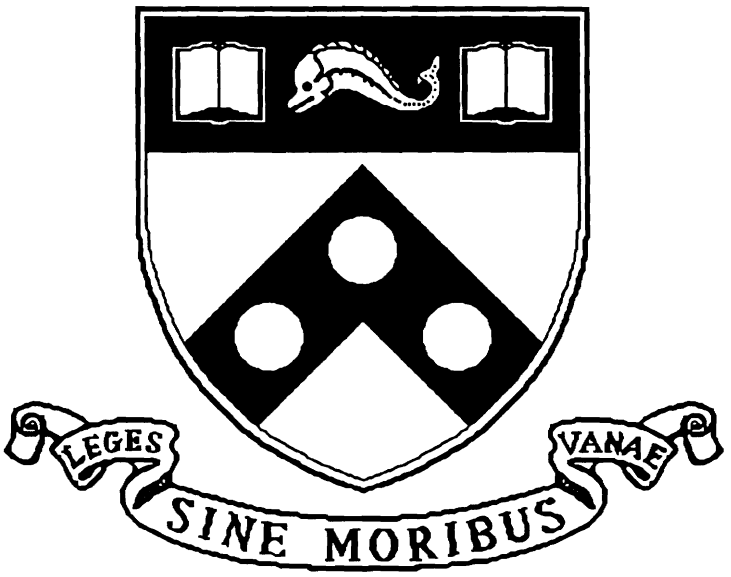
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-37.

The Fast Fourier Transforms As A Database Query

MS-CIS-93-37
LOGIC & COMPUTATION 60

Peter Buneman



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

March 1993

The Fast Fourier Transform as a Database Query

In memory of Oscar Buneman (1913-1993)

Database query languages have a particularly simple structure. The reason for this is partly to make them simple to use, but more importantly to make it possible to optimize them by program transformation. A number of database query languages follow roughly the syntax of Zermelo-Fraenkel set notation:

$$\{\text{Name } x \mid x \in \text{Employee}, \text{Age } x \geq 35\}$$

As a more mathematical example $\{(x,y) \mid (x,z) \in R, (t,y) \in S, z = t\}$ expresses the composition of binary relations R and S. These syntactic forms are called *comprehensions*, and a restricted form of comprehension syntax constitutes relational calculus, the basic query language for relational databases. It has been known for some time that there is an equivalent *algebraic* language, the relational algebra, whose identities serve as the rules for transforming programs. Recent work in which my colleagues and I have been involved[1] has shown that there is a more general algebra underlying comprehension notation. It is more general for two reasons: first it allows us to deal with structures that are not "first order" such as sets of sets; second, it generalizes the application of this syntax beyond sets – to lists and multisets.

A central operation of this algebra is a *mapping* function whose behavior is described by $\vec{f}\{x_1, x_2, \ldots, x_n\} = \{f(x_1), f(x_2), \ldots, f(x_n)\}$. $\vec{f}$ applies f to each member of a set (or list or multiset) $\{x_1, x_2, \ldots, x_n\}$. Another important operation is *flattening*, μ, which, in the case of lists, concatenates a list of lists into a single list. Thus $\mu\{\{1,5\},\{7,3\},\{5\}\} = \{1,5,7,3,5\}$. Again, flattening can be performed on multisets of multisets or on sets of sets, where it is "big" union. When we do not need to be specific about the kind of structure (list, set or multiset) under consideration, I shall use the generic term "collection"

If S is a collection of collections of collections, we have the identity:

$$\mu(\mu(S)) \quad = \quad \mu(\vec{\mu}(S)) \tag{1}$$

In addition, we describe an operation such as summation, Σ, as *well-behaved* because, for any collection of collections of numbers, S, a related identity holds:

$$\Sigma(\mu(S)) \quad = \quad \Sigma(\vec{\Sigma}(S)) \tag{2}$$

Note that Σ is well-behaved for lists and multisets, but *not* for sets.

To see how these operations connect with comprehension syntax, first observe that a comprehension such as $\{e \mid x \in e', y \in e''\}$ can be expressed as $\mu\{\{e \mid y \in e''\} \mid x \in e'\}$, we therefore need only to deal with "simple" comprehensions of the form $\{e \mid x \in e'\}$. In such comprehensions x is a variable that is introduced by the binding $x \in e'$ and may occur in e. Simple comprehensions may be eliminated with the identity:

$$\{e \mid x \in e'\} \quad = \quad \vec{f}(e') \quad \text{where} \quad f(x) = e \tag{3}$$

[1]See P. Wadler. Comprehending Monads. *Proc. ACM Conf. on Lisp and Functional Programming*, Nice, June 1990. See also V. Breazu-Tannen, O.P. Buneman and L. Wong. Naturally Embedded Query Languages. In *Proc. Int. Conference on Database Theory*, Berlin, October 1992. Springer-Verlag LNCS, pages 140-154. I am grateful to Val Breazu-Tannen for his careful reading of this note.

Repeated applications of (3) can be used to eliminate completely comprehension syntax from an expression. Also from (3), the following identities may be derived:

$$\{f(e) \mid x \in e'\} = \vec{f}\{e \mid x \in e'\} = \{f(y) \mid y \in \{e \mid x \in e'\}\} \quad \text{if } x \text{ does not occur in } f \tag{4}$$

A natural question to ask is whether arrays fit into this scheme of collections. To be consistent with what we have already developed we must adopt the convention of many programming languages that arrays are one-dimensional, and that a two-dimensional array is represented by an array of arrays. Let us use the notation m-array to describe an array of length m and $m \times n$-array to describe an array of length m of arrays of length n. Our comprehension syntax appears to make sense for arrays $\{2x \mid x_i \in V\}$ will result in array whose ith component is double the ith component of V, and $\{ix \mid x_i \in V\}$ will result in an array whose ith component is i times the ith component of V. Note the deviation from mathematical convention, in which one would write $\{ix_i \mid x_i \in V\}$. In comprehension notation, the binding $x_i \in V$ is taken to bind both the variable x and the variable i to each successive value–index pair in V.

As we progress from sets to multisets to lists, we have increasing structure. For example, lists may be reversed, but reversing is meaningless for sets or multisets. Arrays appear to be a further step in this progression, but what are the additional operations and identities? I must confess to expedience here; I do not know the complete answer to this question. However there are two operations that bear close relationship to what we have already developed. Borrowing symbols from APL[2], ϕ transposes an $m \times n$-array to produce an $n \times m$-array. We have the associated identity:

$$\{\{e \mid x \in e'\} \mid y \in e''\} = \phi\{\{e \mid y \in e''\} \mid x \in e'\} \quad \text{provided } x \text{ does not occur in } e'' \text{ nor } y \text{ in } e' \tag{5}$$

A second operation is "reshape" $\rho(m, n)$. If V is an mn-array, $\rho(m, n)V$ is an $m \times n$-array. Associated with this operation, we have the identity:

$$\{e \mid x_i \in V\} = \mu\{\{e' \mid x_b \in W\} \mid W_a \in \rho(n, m)V\} \quad \text{where } e' \text{ is obtained from } e \text{ by substituting } ma + b \text{ for } i \tag{6}$$

The m-shuffle operation needed in the fast Fourier transform and many other algorithms is simply defined as $\mu(\phi(\rho(m, n)V))$ for an mn-array V.

Lastly, we need to express the fact that we can interchange the order of summations over an $m \times n$-array. If W does not occur in e then

$$\Sigma\{\Sigma\{e \mid x_b \in W\} \mid W_a \in V\} = \Sigma\{\Sigma\{e' \mid x_a \in Y\} \mid Y_b \in \phi(V)\} \quad \text{where } e' \text{ is obtained from } e \text{ by interchanging } a \text{ and } b \tag{7}$$

Using comprehensions, we can implement the *discrete Fourier transform* (DFT) of an mn-array V as

$$\Phi^{mn}(V) = \{\Sigma\{\omega_{mn}^{ij} x \mid x_j \in V\} \mid i \in I^{(mn)}\} \tag{8}$$

(I have omitted a sign and a factor of $\sqrt{2\pi}$). In this, ω_n^i is the ith power of the nth principal root of 1, and $I^{(n)}$ is the array $0, 1, \ldots, n-1$. It is important to stress that (8) is a *program*. If you will

[2]K. Iverson. *A Programming Language*, Wiley, 1962.

allow the expansion of Greek letters into the kinds of symbols used in conventional languages and an explicit notation for multiplication and subscripting, it is a program that can be written in certain existing functional programming languages and database query languages. In such languages, the program would have a running time proportional to $(mn)^2$.

Using (6) and (7), we can transform the DFT (8) into

$$\{\Sigma\{\Sigma\{\omega_{mn}^{i(mb+a)}y \mid y_b \in z\} \mid z_a \in \phi(\rho(m,n)V)\} \mid i \in I^{(mn)}\}$$

Writing W for $\phi(\rho(m,n)V)$ and using (6) gives us

$$\mu\{\{\Sigma\{\Sigma\{\omega_{mn}^{(nd+c)(mb+a)}y \mid y_b \in z\} \mid z_a \in W\} \mid c \in I^{(n)}\} \mid d \in I^{(m)}\}$$

Moving a product over a sum (I did not specify this obvious transformation) gives us:

$$\begin{aligned}
& \mu\{\{\Sigma\{\omega_{mn}^{a(nd+c)}\Sigma\{\omega_n^{cb}y \mid y_b \in z\} \mid z_a \in W\} \mid c \in I^{(n)}\} \mid d \in I^{(m)}\} \\
= \; & \mu\{\vec{\Sigma}\{\{\omega_{mn}^{a(nd+c)}\Sigma\{\omega_n^{cb}y \mid y_b \in z\} \mid z_a \in W\} \mid c \in I^{(n)}\} \mid d \in I^{(m)}\} \quad \text{by (4)} \\
= \; & \mu\{\vec{\Sigma}(\phi\{\{\omega_{mn}^{a(nd+c)}\Sigma\{\omega_n^{cb}y \mid y_b \in z\} \mid c \in I^{(n)}\} \mid z_a \in W\}) \mid d \in I^{(m)}\} \quad \text{by (5)} \\
= \; & \mu\{\vec{\Sigma}(\phi\{\{\omega_{mn}^{a(nd+c)}t \mid t_c \in \{\Sigma\{\omega_n^{cb}y \mid y_b \in z\} \mid c \in I^{(n)}\}\} \mid z_a \in W\}) \mid d \in I^{(m)}\} \quad \text{by (4)} \\
= \; & \mu\{\vec{\Sigma}(\phi\{\{\omega_{mn}^{a(nd+c)}t \mid t_c \in \Phi^n(z)\} \mid z_a \in W\}) \mid d \in I^{(m)}\} \qquad (9)
\end{aligned}$$

This last expression is a program that implements the DFT of size mn using m DFTs of size n. Having computed the latter, the number of additional operations used in (9) is readily seen to be proportional to m^2n. Thus if $T(n)$ is the time needed to compute the DFT of n, this transformation shows that $T(mn) = mT(n) + Km^2n$ for some constant K. When $m = 2$ we have the well-known recurrence relation $T(2n) = 2T(n) + K'n$, which shows that the number of operations needed to evaluate the discrete Fourier transform of an array of length n is proportional to $n \log_2 n$.

What has this bought us? There is some interest in computer science in finding an efficient implementation from an initial program by direct manipulation of the source code and, at the same time, establishing the correctness of that implementation. Another point is that databases and database languages are often criticized for their inability to maintain and manipulate scientific data. I hope that the foregoing indicates that it may be possibile to correct this.

Of course, the program I have developed here is a very, very long way from the tight and elegant code that my father once showed me in his implementation of the fast Fourier transform[3]. However, I believe he would have taken pleasure from the apparent connection between our two subjects.

Peter Buneman
University of Pennsylvania
March, 1993

[3]O. Buneman, In-situ bit-reversal ordering for Hartley transforms, *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-37, pp. 577-580, 1989; also, patent 4,878,187 "Apparatus and Method for Generating a Sequence for Sines and Cosines."